# Section: Introduction to Puppet

Puppet is a configuration management tool, you use it to keep the configuration of your machines (called nodes) consistent.

There are several terms used in a Puppet installation, we'll introduce those here.

Nodes, are the machines that are configured by your puppet installation.

Facts are things which can be known about a Node. Things such as CPU, memory, ipaddress, operatingsystem. facts are retrieved using the facter command, which we will show now

# facter |less

The Master or puppetmaster, this is the machine that your nodes communicate with for their configuration instructions. In recent versions of Puppet, the puppetmaster runs a Java server process named puppetserver.

PuppetDB is the service which Puppet uses as a data-warehouse of information about nodes.

All communication between the nodes and the master is done via TLS encryption. The encryption keys are created on the certificate authority host or CA or PuppetCA. The PuppetCA acts as the certificate authority for your puppet installation, all the nodes and your puppet masters will have their certificates signed by the CA.

Another term you may hear in large installations is MoM or Master of Masters. In a large installation, puppet is used to configure the puppetmaster servers in a multilevel fashion. This is known as having a master that is the master for your other masters.

Puppet is also a language. Puppet is known as a DSL, a domain specific language. You can think of it like another language similar to Python and Ruby. Puppet is currently written in Ruby, but since it is a DSL, it is not limited to being implemented in Ruby.

Puppet language is made up of control structures and resources. Control structures are similar to those seen in many other languages, conditionals such as if and case.

Resources are made of up types and attributes. A type in Puppet is something like a package, service or file. There are several types provided by Puppet. Each type has a number of attributes. Attributes are the specifics of the type, for instance the following is a file resource:

file { 'hello':

path => '/tmp/hello',

content => "Hello World!\n",

}

In this example, the type of the resource is File, file has several attributes. The complete list is available at the type reference page on Puppet's website (URL).

https://docs.puppet.com/puppet/latest/reference/type.html

Our example has a path, the path to the file. Contents, the contents of the file.

This resource will create a file in the /tmp directory named hello with the contents "Hello World!".

To apply this file to a node, we can use the puppet apply command. Puppet apply is used to apply code directly to a node without using a puppetmaster.

To do this we will create a manifest, a manifest is how puppet organises resources. Resources are organized into manifests. Manifests are then applied to nodes.

In this example we will create a manifest named hello.pp (manifest files should have the .pp extension).

hello.pp

notify { "Hello World!\n": }

Putting our code into hello.pp we can apply this to a test machine doing

# puppet apply hello.pp

[while puppet is running]

Puppet will now compile our manifest into a Catalog. A catalog is how puppet organises all the manifests that are to be applied to a node.

metaparameters - metaparameters are how puppet handles dependencies. Initially these are the hardest part of puppet for most new puppeteers to understand, the four main metaparameters are before, require, notify and subscribe.

before and require are simple ordering parameters

if we have the following two resources, a file and a directory, we can put them in our manifest like this.

file {'first':

path => '/tmp/order/first',

content => 'I'm first',

}

file {'order':

path => '/tmp/order',

ensure => 'directory'

}

Next up: Modules, include, contain, and metaparameters

Resources can be organized into classes. Classes are logical groupings of resources, classes can be addressed as subclasses of another class. In the following example we have a class and a subclass.

class example {

file { 'hello':

path => '/tmp/hello',

contents => "Hello World!\n",

}

}

class example::goodbye {

file { 'goodbye':

path => '/tmp/bye',

contents => "Good-bye Hello!\n",

}

}

To apply these two classes to a node, we can use the include syntax. Include is used to include a class.

[add an include line and then show them puppet apply again.]

puppet apply example.pp

# Section: code

Main bulletpoints for this section: Modules, modulepath, environments, environment.conf, git, control repository, r10k, puppetfile

## Subsection: Templates and files

Create another subclass that creates a file and another that uses a template.

example::file

# example::file

class example::file {

file { 'source-file':

path => '/tmp/example-file',

source => 'puppet:///modules/example/example-file',

}

}

example::template

This is an example template

Facts are available using @

This nodes hostname is <%= @hostname %>

This node has the following interfaces:

<% @interfaces.split(',').each do |interface| -%>

<%= interface %>

<% end -%>

Done.

class example::template {

file {'example-template':

path => '/tmp/example-template',

content => template('example/example-template.erb')

}

}

## puppetserver

We'll start by using puppetserver instead of puppet apply. When we are using puppetserver, we need to make sure our node has signed certificates. This isn't an issue here, since I'm going to show you these examples on the master.

First thing is to ensure that puppetserver is running.

systemctl status puppetserver.service.

verify with lsof -i :8140

We can run puppet against our node

## Environments / environment.conf

by default nodes run in the production environment

A quick word about environments. Environments are how puppet decides which code to apply to your nodes. Each node has an environment assigned to it, the default environment is production and that corresponds to the /etc/puppetlabs/code/environments/production directory. [go to that directory while speaking] This directory lives on your puppetmaster or master.

In the environment directory we can have directory name modules which will be searched for modules. We can also have an environment.conf file [our directory will have one].

The environment.conf is used to specify an alternate modulepath per environment. So that you can have two different environments with different modules and environment.conf files and have different behaviour for your agents.

# control repository git and r10k

Now we'll show how to configure a control repository, or control\_repo, which is used to organize your modules. It is a common practice to have a git repository be in charge of the contents of your environments directories. We will make a control repository and upload it to github. This repository will have an environment.conf file and a site manifest (site.pp). This is a very simple example.

cd /srv/git/repos

git init --bare control.git

cd

git clone /srv/git/repos/control.git

cd /srv/git/repos/control

git status

git branch production

git checkout production

vim environment.conf

modulepath = modules:$basemodulepath

manifest = manifests/site.pp

vim site.pp

node default {

notify {"This is production": }

include example

}

git branch development

git checkout development

vim site.pp

change production to development.

now manually go and checkout the different branches in /etc/puppetlabs/code/environments

I want production and development to be there.

create repository, put environment.conf file in there, then site.pp with something simple like a notify.

I need a puppet server and a node.

Then make a different branch, show how those are mapped to the environments.

show how to checkout the repositories locally first maybe, then keep r10k section completely separate.

run the agent, show different results.

Stop make r10k a new section.

# r10k.

configure r10k to use this git repository, maybe make the repo local to the puppetserver to start

/etc/puppetlabs/r10k/r10k.yaml

---

:sources:

:control:

remote: '/srv/git/repos/control.git'

basedir: '/etc/puppetlabs/code/environments'

prefix: false

so we have the branches from before, now we can show how r10k can do the deployment of those branches for us quickly.

Now create another git repository, call it something else like mpli

make branches in mpli

:mpli:

remote: '/srv/git/repos/mpli.git'

basedir:''

preifx: true

what the prefix setting actually means, just talk about it, show it in action with just the single repository

clean out the environment directory, run r10k to have it create the directories using r10k deploy environment -p

Another tools, librarian-puppet came before r10k, it introduced the Puppetfile, a file which contains references to modules. Using a Puppetfile, r10k can deploy modules within your environment directory. In the modules directory. So for our earlier example, we can create a Puppetfile with the following contents:

mod 'puppetlabs/ntp', '4.1.0'  
mod 'puppetlabs/stdlib'

talk about the forge here.

:put:file:here

mod 'something',

git: => '',

ref: => ''

mod 'somethingelse',

git: => ''.

ref: => ''

Now when we run r10k deploy environment -p (the -p means use the Puppetfile)

We see that the modules are placed into the modules directory.

Hint at how you can get git to deploy these changes with r10k in the git hook.